Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 271 (2011) 41–62

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Testing Data Consistency of Data-Intensive Applications Using QuickCheck

Laura M. Castro[1,2]

Department of Computer Science
University of A Coruña
A Coruña, Spain

Thomas Arts[3]

Computer Science and Engineering
Chalmers University
Göteborg, Sweden

Abstract

Many software systems are data-intensive and use a data management systems for data storage, such as Relational Database Management Systems (RDBMS). RDBMSs are used to store information in a structured manner, and to define several types of constraints on the data, to maintain basic consistency. The RDBMSs are mature, well tested, software products that one can trust to reliably store data and keep it consistent within the defined constraints.
There are, however, scenarios in which passing the responsibility of consistency enforcement to the RDBMS is not convenient, or simply not possible. In such cases, the alternative is to have that responsibility at the business logic level of the system. Hence, from the point of view of testing data-intensive applications, one of the most relevant aspects is to ensure correctness of the business logic in terms of data consistency.
In this article, we show how QuickCheck, a tool for random testing against specifications, can be used to test the business logic of an application to increase confidence on data integrity. We build an abstract model of the data containing the minimum information necessary to create meaningful test cases, while keeping its state substantially smaller than the data in the complete database. From the abstract model we automatically generate and execute test cases which check that data constraints are preserved.

Keywords: Software verification, Software testing, Model Based Testing, Software Tools, QuickCheck.

doi:10.1016/j.entcs.2011.02.010

1 Introduction

Many applications use one or more databases to store vast amounts of data. They also provide a number of different interfaces to inspect and modify the data. Some application interfaces may be accessed from a web interface, others may be used from a desktop application. Different users have different access rights, and therefore are presented with different interfaces to the data.

The database normally constrains certain relations on the data that cannot be violated; given that the database management system is correctly implemented. On top of these built-in constraints there are other rules that are imposed by the application. These constraints involve not only data format or relationships, but also non-trivial calculations. For example, constraints on top of built-in constraints could be to ensure that a zip code is of the right format for a given country, or that a user can only update certain profile information after having completed updating some registration data. In general, these constraints are verified by the application, that is, the system validates that input data is of the right format, or that a process is followed before the database is actually queried. Neglecting to verify the information, will result in storing data in the database that does not fulfil the constraint.

There are additional reasons for the need of constraints which are unenforceable by the database. For example when the constraint is time-dependent, or has performance and efficiency demands that cannot be guaranteed by the database itself. Commonly, many constraints are too business-specific, and therefore there is no need nor desire to "hard-wire" them in the database, in particular not if the database is to be shared by other systems. For these reasons, it is the *business logic* of the application which needs to cope with these business-specific constraints, also called *business rules* [5].

When testing a data-intensive application, it is crucial to provide enough confidence in that the system cannot reach a situation where a business rule is violated and data is left in an inconsistent state. This should be tested independently whether such is a consequence of one or several invocations of interface functions with unexpected input or unconsidered scenarios.

Previous attempts have faced the challenge of business rules definition and modelling, either by proposing new development methodologies, or by formulating modifications to existing ones [5, 8]. But few efforts have been devoted to checking the actual enforcement of business rules. To the best of

[1] This research was partly sponsored by three grants: FARMHANDS (MEyC TIN2005-08986 and XUGA PGIDIT07TIC005105PR), EU FP7 Collaborative project *ProTest* (grant number 215868), and AMBITIIONS (MICIN TIN2010-20959).

[2] Email: lcastro@udc.es

[3] Email: arts@chalmers.se

our knowledge, the most practical approach to this matter is the one presented in [16], which proposes an extension to JUnit [14] to perform better tests, data-oriented according to business rules testing. Other testing approaches have focused on code coverage and database-queries correctness [9, 12], and thus can be considered as orthogonal to our work.

We propose the following approach: In order to ensure that the business rules are respected, we formulate them as invariants over the data in the database. We check these invariants on the database after random invocations of the application interface functions. If any other of the interface function calls allows the database to enter a state in which an invariant is violated, we have detected an error in the application. In order to generate meaningful tests, i.e., to control test randomness in a smart way, we create an abstract model of the database. For example, if we enter a certain item in the database then we need to remember its key or identifier to perform subsequent operations related to the same item. In order to keep the model simple, it is important to keep it abstract and store as little information as possible in the model state, definitely less than the data stored in the database. If not, modifying the model state after an operation would involve the same complexity as modifying the database. As such, we would copy a large part of the business logic operations in the model, which we want to avoid.

We formulate the model as a QuickCheck [4] specification [4, 13]. QuickCheck is a tool for guided random testing. QuickCheck specifications are written in the functional language Erlang [2, 7] with some QuickCheck specific macro definitions. The tool automatically generates and executes test cases from the provided specifications.

The method for testing data-intensive systems by using QuickCheck that we propose, is presented in this paper by means of a little example illustrating an on-line shop. The method was developed, however, while testing a real insurance system [1]. We started testing that real application after it had been in service for a few years in order to obtain knowledge on how to test such systems. We have not identified any faults in the production code by using our method, but we were able to show usability of the method by injecting faults in the production code and detecting them by running the tests originating from our method. Moreover, after developing the method, a master student project was conducted in which another commercial database intensive system was tested in the same way. That project resulted in the detection of a number of faults [15] and showed that the method was applicable to a broader domain.

The paper is structured as follows: In Section 2 we explain the concept of business rules and we introduce our leading example, an on-line shop, to

[4] We use Quviq QuickCheck version 1.19, a commercially available version of QuickCheck.

clarify this concept. The first step of our method for testing database systems is to formulate the business rules as SQL queries. In Section 3 we describe the kind of sequences of interface calls that we need as test cases for testing our business rules. In Section 4 we present the model that is used to automatically generate those sequences. The model is described as a finite state model in the specification formalism used by QuickCheck. In Section 5 we describe how we have evaluated the presented method in real industrial examples; demonstrating that it is a feasible and effective method to find faults that can cause violation of the business rules. We conclude in Section 6 by summarising the method we contribute with.

2 Business Rules

In this section we introduce a simple application that serves as the running example in the paper. The example demonstrates how in practise certain data constraints are better defined in the business logic layer instead of in the persistence layer of a system. We describe a general method on testing data consistency that generalises to much larger databases, such as the ones used in the case studies we will see in Section 5. The method is used for testing that the application cannot violate the business rules by accessing the database through the business logic layer, i.e., by using any of the possible application interfaces. The method should be applicable to relational as well as non-relational databases with several concurrent clients accessing the system. However, we have not explicitly addressed distributed databases or time critical databases in our research.

2.1 Leading example: an on-line shop

We borrow a simple example from Willmor and Embury [16]. We implement an application to deal with customers, products and orders, in which we introduce a *status* for customers, either 'gold' or 'non-gold'. The meaning behind this status is that only gold members can purchase some special products. An Entity-Relationship diagram for this example is shown on Figure 1.

When translating this diagram into a relational database schema, a set of constraints appear, that will be implemented as database constraints (entity keys as primary keys, one-to-many and many-to-many relationships as foreign keys, etc.). However, constraints like *when* a customer acquires 'gold' status, or *which products* are only to be purchased by 'gold' customers, are the kind of constraints that are not desirable to be implemented at the database level. Not only because these constraints may vary during system life or operation (the list of 'featured products' can be different each month or week, a

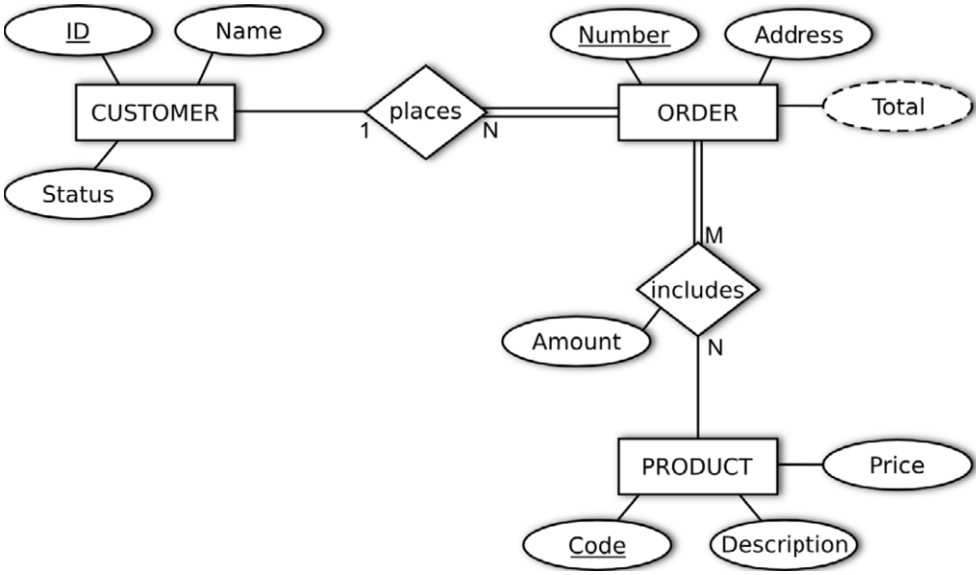

Fig. 1. Simple ER diagram example

'gold' membership can be obtained by purchasing a number of orders or just be granted temporarily and selectively to incentive consumer shopping), but most importantly, it is a domain-level property which is not data-intrinsic. Thus, these constraints are considered business rules and are taken care of at application business logic level.

2.2 Invariants as SQL queries

As stated in [16], correct implementation of business rules by a system can be tested by translating each rule into an SQL query which can be evaluated against the database. For instance, the following SQL sentence will check that customers without 'gold' status cannot have an order placed that includes 'gold' featured products. The corresponding business rule is violated if anything else than the empty set is returned:

```
SELECT customer.id
  FROM customer, order, order_products
 WHERE customer.id  = order.customer_id
   AND order.number = order_products.ord_nmb
   AND customer.status <> 'gold'
   AND order_products.product_code
    IN <featured product list>
```

2.3 Violation of business rules

Assume gold membership is to be obtained when the user has at least five orders placed. Assume also that available interface functions in the system are the addition, update and removal of entities (customers, products, orders). Then, there are many ways in which the previous business rule can be broken. Not only single operations like a non-gold user placing a new order including a featured product will violate the constraint above, the following sequence will do as well:

(i) non-gold customer X places fifth order (system updates status to 'gold')
(ii) same customer X (now 'gold') places order for a featured product
(iii) customer X cancels one of the first five orders

There are different system behaviours that can be implemented to avoid reaching the inconsistent situation that this sequence will lead the database to (according to business rules). For instance, The application can prevent a gold customer from cancelling an order if that would remove his/her gold status, or else cancelling such order could not only remove the gold membership condition, but also automatically cancel the order that included the featured product. When testing data-intensive applications for data consistency, we do not really care about which policy is actually implemented: we aim to ensure that, whichever it is, it guarantees business rule compliance at all times.

3 Test sequences

From the previous example we conclude also that data-intensive application tests must include sequences of calls to interface functions. Straight violations of business rules will usually be taken care of by the developers when implementing the business logic (i.e., a non-gold customer will never be allowed to place an order including a special product). On the contrary, uncommon or a-typical sequences of calls may very well potentially cause data inconsistencies, and are more likely to be overlooked. Besides, in real applications, it is impractical to generate a fair number of combinations of interface calls manually. Hence, we use a tool to help generating sequences of calls in order to test the business logic.

3.1 Meaningful test sequences

To get around any unconscious presumptions and avoid omitting relevant scenarios or possibilities in our test sequences, automatic test generation tools that provide random input can be helpful. However, completely randomising

a test sequence will generate lots of meaningless tests, e.g., repeatedly trying to add non-existing products to an order, or cancelling unknown orders. In purely random generation, we have an extremely small likelihood of generating a meaningful test like the one involving the same customer performing three different operations. Therefore, we need to use guided random generation. This is were we introduce a model of the state of the database.

The business logic of our example provides a number of interface functions, for example, to register a new customer (`new_customer`) or to disable a certain product for sale (`delete_product`). It also contains more complex functions such as registering that a certain customer placed a certain order (`place_order`). The safest assumption for testing business logic is to accept that any function defined and exported for public use can be called in arbitrary sequence with arbitrary arguments. However, as we previously mentioned, we want to get as many meaningful tests as possible; therefore we want to use knowledge on which interface calls have been previously invoked and which results they had, in order to generate the next interface call in a testing sequence.

We write our testing sequences as data structures in the language Erlang. These data structures are almost test programs, but instead of programs we generate a data structure that we interpret as a program. The reason for doing so is that it allows us to treat the test case as an object in the test language. Therefore, we can simplify or modify the test case in any way we like; which helps if we want QuickCheck to find a simpler test case after having found a failing test case.

The following example shows how such a test sequence looks like. The sequence represents the failing scenario described in Section 2.3. Variables are written with an uppercase character in Erlang and words starting in lowercase are atoms, which can be considered constants. Tuples are contained in `{ }` and lists in `[ ]`.

```
Id   = {call,business_logic,new_customer,["Laura"]},
Nr1  = {call,business_logic,place_order, [Id,1277]},
Nr2  = {call,business_logic,place_order, [Id,7027]},
Nr3  = {call,business_logic,place_order, [Id,3112]},
Nr4  = {call,business_logic,place_order, [Id,4983]},
Gold = {call,business_logic,place_order, [Id,9002]},
{call, business_logic, cancel_order, [Nr3]}
```

The tuples with first argument `call` are interpreted as interface calls, the first tuple, for example, would call the unary function `new_customer` in the module `business_logic` with the string `"Laura"` as argument.

Note that we need a return value from the first call in consecutive calls,

and we need to remember the returned order numbers in order to cancel one (last step of the sequence). We expect this sequence to be a valid scenario, i.e. a sequence of calls that indeed could be made from a user interface.

There are sequences that we could expect to be prevented, such as creating a new customer and directly ordering a featured product (for which gold status is needed). A user interface may disable the opportunity to order featured products if the user has not reached the gold status yet. However, we do not want to make any assumptions in this sense, so we allow the generation of such sequences as well.

3.2 Positive and Negative Testing

We call an interface function call a *positive test* if it is expected to return successfully, we call it a *negative test* if it is expected to return an error. Whether or not we expect an interface function to return successfully or to return an error depends on the state of the database or, in other words, on the preceding interface calls. For example, some tests would violate a business rule when performed at a given state, so we test that, given such state of the model, the result is an error message by the interface and the database state does not violate the business rules afterwards.

Hence, a test fails if the application crashes during consecutive calls, when the result of one of the interface functions is unexpected with respect to our model, or when the database is left in a state that violates the business rules.

As an example, in our on-line shop, an error may be a perfectly valid result for the `cancel_order` call (the last step in our sample test sequence), if the business logic does not allow order cancellation conflicting with the gold status update. Similarly, if the business logic is chosen differently, the cancellation operation may be successful, provided that the gold-status is revoked as a result of it.

4 Test model

To be able to automatically generate meaningful both positive and negative tests, we need to keep a minimum *test state*, that will include the least amount of data needed to generate related interface calls in our test sequence. For example, such state would need to hold the customer ID of a customer if we would like to perform consecutive operations that involved that same customer. The state would function as a kind of 'history', such that we know what customers, products, etc. have taken part in former function calls without the need to inspect the database (or trusting that they have been properly

stored in it as part of the previous operations). In other words, we want to model system behaviour and use *model based testing*.

Another requirement is that we would like to be able to automatically obtain simpler test cases when a test case fails; this helps in the fault analysis and debugging process. Thus, if a test fails, then we would like to obtain a sequence that in all possible ways has been made shorter, and for which all arguments of the called functions have been simplified. QuickCheck is able to perform these manipulations automatically, provided that we present the test case as the special data structure explained above. QuickCheck can modify the sequence, but it needs to preserve the test case as meaningful. Simply removing lines from the sequence, for example, might result in inappropriate or just irrelevant tests.

QuickCheck has a number of specific libraries for representing state machine models and we use one of them (`eqc_statem`) to model our data-constraint business rules test cases.

Note that we use QuickCheck to our advantage, but the basic method of testing data-intensive applications could also be implemented by using different tools for test case generation and execution.

4.1 Generating Random Commands

To specify the system under test, we create a state machine. Ultimately, the core of a data-intensive system is the database behind it, thus we can think about it as a stateful system, where the 'state' is the data stored in the database at any particular moment. Using a state machine we can specify how we can bring the system from one state into another.

Now the challenge is to abstract from the data in the database and use that abstraction as a test model. Otherwise, we would end up with a copy of the entire database as state. Not only would that be potentially too large to handle, more seriously, we would have to re-implement (at least part of) the business logic to deal with it, since re-using the implementation under test would not enable us to find errors. Of course, implementing software twice is an unattractive idea; apart from the work involved, one would probably make similar mistakes.

Our approach is to make sure we have just enough data in our state to bring the system in all kind of different states.

We start by identifying the interface functions in our application. For example, in our shop example we have: `new_customer`, `add_product`, `place_order`, etc. Of course, in a small example like the one we present here, the interface functions are limited and simple, but it explains our point.

We want to create a sequence of interface calls, or *commands*, where each command has a certain likelihood to appear in the sequence. We use QuickCheck's `command` generator for that in combination with the `frequency` combinator. The command generator takes the test state as input, and commands are generated relative to the present state. The frequency combinator takes a list of *tuples* as input, the first argument in the tuple being the relative weight with which that command will be generated. The possibility that an alternative is chosen is the frequency that is assigned to it divided by the total of all alternatives; a frequency of zero means the alternative is never chosen.

```
command(S) ->
 frequency([
  {1,{call,customer_facade,new_customer,[name()]}},
  {4,{call,product_facade,add_product,[...]}},
  {2,{call,order_facade,place_order,[...]}}, ...]).
```

The code above specifies that given abstract state `S`, specified commands are generated with frequencies 1, 4, 2... out of the total number of listed commands. Thus, the second alternative is chosen more than half the time (4/7) with respect to the other alternatives.

Commands are tuples with four elements:

(i) the tag `call` specifying that this is a function call;

(ii) the name of the module where the function is found;

(iii) the name of the invoked function;

(iv) a list with its arguments.

The module and interface functions are statically known; the frequencies in which they should be used are domain knowledge of the test specifier. Which frequencies one wants to use depends on what one would like to test. In the general case, one would probably like each interface function to occur equally often. However, in our case, we would like emphasis the tests more on adding products and placing orders for those products. By giving more weight to the addition of product than to the placement of an order, we achieve that the placed orders contain on average at least a few products. Had we given them the same frequency, then many tests would probably test placing orders without selected products or with one product selected. We are also not that interested in tests that add many different users, more in tests with a few users that all order many products. This is achieved by reducing the possibility to create new users.

With the use of QuickCheck one can then automatically generate and run as many different tests as one like. One should make sure to instruct

QuickCheck to measure how often the same person on average places an order, how often empty orders are placed, how the distribution of products per order looks like, etc. by using QuickCheck's `collect` and `measure` utility functions. By adjusting the frequencies, one may obtain a better test case distribution. If that cannot be achieved, one can always define two different distributions and run a bunch of test cases with one distribution and another bunch of test cases with another distribution. The good thing is that statistics can be easily obtained about what has been tested.

4.2 Generating random data

The remaining task is to generate arguments that make sense for each of the specified commands. It is easy to specify a generator for the argument of `new_customer`; in our simplistic example we only need a `name` but even if we provide address, email address, telephone number, etc. we can generate random data for these fields. We obtain the type of the data from the table definition and can automatically transfer that in a data generator for QuickCheck. For example, when the SQL definitions of the customer database table is:

```
CREATE TABLE customer
   (id INTEGER CONSTRAINT cust_prk PRIMARY KEY,
    name VARCHAR,
    status VARCHAR CONSTRAINT cust_nn NOT NULL)
```

Then we can use the generator defined by

```
name() ->
   varchar().
```

where `varchar` is a QuickCheck generator that creates random lists of appropriate characters.

We could implement a more specific generator, e.g. creating the name as a firstname and a lastname both starting with an uppercase character and the rest in lowercase. That, however, would only be significant if this indeed is part of the business rules and in case the interface functions should ensure this. Keep in mind that this allows to add the same user twice. We want this to be a potential test case, since the business logic may prevent or allow that situation.

Generators specification gets trickier when we want to call the function `place_order`, since we need to provide a customer ID and a product code. The problem with generating a random customer ID and random product code is that they will most likely not be in the database. In the intentional

testing approach [16] the product and customer are automatically generated when not present in the database. We propose a different tactic and use the test state (`S`) to keep track of created customers and products in order to use them in calls to `place_order` and similar. In addition, we still choose now and then, with a small probability, a customer or product (or both) that is not present in the state, to test correct handling of those error scenarios (negative testing).

4.3 Data as part of the model state

To manage the state machine state in our example, we define a record data structure with two fields, each one containing a list: a list of customer identifiers, and a list of product codes. Initially all lists are empty:

```
initial_state() -> #shop{customers = [], products = []}.
```

The lists of products will later be inhabited by codes of products that we have added in the executed test. Thus, the field `products` stores a list of indices. Respectively, the customer list contains a list of the customer data type, which is a record with two fields, `#customer{id, orders=[]}`, the customer identifier `id` and a list of `orders` placed, where an order is stored by its order identifier, needed for cancelling the order. We could also add all product codes that are ordered in the order information, but since our approach is to keep the model as simple as possible, therewith reducing the amount of work necessary, there is really no reason to do so. We only store the data that is created by the database and not part of the data provided in the test cases.

Each interface function execution may have an influence on the state, so we define a state transition function `next_state` in our state machine model to reflect that. This function takes three arguments, first the present abstract state `S`, then a place holder for the result that the interface function returns, and the representation of the interface function. The `next_state` function returns the updated test state, which is an abstraction of the state we expect the database to be in after executing the interface call.

```
next_state(S, Id, {call,customer_facade,new_customer,[Name]})->
 NewCustomer=#customer{id = Id, orders = []},
 UpdatedCustomers=[NewCustomer|S#shop.customers],
 S#shop{customers=UpdatedCustomers};
```

In this function clause, a new customer record is created, and the field `Id` of a new `customer` (`NewCustomer`) records is initiated with the value that the function will return when executed. The `orders` field is initiated with

the empty list. After that, the new customer is added to the list of already existing customers (`S#shop.customer`) and the state `S` is updated by replacing the value of the field `customers` by this new list of customers.

This `initial_state` and `next_state` functions allow us to generate a huge amount of different commands that refer to expected return values. In order to add some randomness too, we complete the command generator as follows:

```
command(S) ->
 frequency([
  {1,{call,customer_facade,new_customer,[name()]}},
  {4,{call,product_facade,add_product,[pr_code(S)]}},
  {2,{call,order_facade,place_order,[cst_id(S),pr_code(S)]}}
   ...]).

cst_id(S)->
  oneof([integer()]++[keys(S#shop.customers)]).

pr_code(S)->
  oneof([integer()]++[S#shop.products]).
```

where the generators `customer_id` and `product_code` pick a random integer (since entity keys are represented by integers in the database of the on-line shop), and add it to the list of known keys. Besides, the QuickCheck generator `oneof` takes a random element from a given list [5]. Thus, we can generate calls to functions such as `add_product` or `place_order` with identifiers of products and/or customers that may already exist or not (with a lower probability) as arguments.

4.4 Validating the result of a test

So we have developed a state machine that can be used to generate arbitrary sequences of interface calls that relate to each other, allowing both positive and negative testing. However, so far we can only generate test cases, and even run them, but we still need a way of validating the results of the calls. There are two things to validate, first that the calls themselves return an expected value, and second that the database state remains consistent with the business rules.

If we assume for a moment that we have successfully executed a generated sequence of interface calls, we would like to know whether the business rules have been violated. In fact, one probably would like to check this after each

[5] The function `keys` extracts the keys from the `customer` records.

executed command separately, but we propose to check it only at the end of the testing sequence. Our assumption is that if the database is brought in a state that is inconsistent with the set of business rules, then the consecutive invocation of interface functions will be unlikely to repair it. Moreover, we run thousands of test sequences, and if the business logic can be violated with a sequence that repairs the violation afterwards, then we most likely also run, as one of the other tests, a shorter sequence that violates the logic and does not happen to fix it later. The advantage of checking violation of the business logic only once per testing sequence is that it makes testing faster and hence we can run more tests in the same time.

In order to test the status of the database after-test with our business logic constraints, we translate each business rule into a database check and include all these as part of an invariant function which is executed after the generated sequence. Each business rule is embedded in a database transaction consisting of one or more SQL queries. For example, the invariant for our running example is:

```
invariant() ->
 {ok,Connection}=db_interface:start_transaction(),
 Result=business_rule(Connection),
 db_interface:rollback_transaction(Connection),
 Result.

business_rule(Connection) ->
 [] == db_interface:process_query(Connection,
   "SELECT id "
   " FROM order_products NATURAL JOIN customer "
   " WHERE status <> 'gold' "
   " AND code IN ( "
   ++str:join([to_list(P)||P<-?GOLDEN],",")++
   " )").
```

Since we use a transactional database, each access to the database is performed as a transaction. In order to be sure that the invariant checking does not affect the database state, we undo the transaction immediately after performing it. Strictly speaking there is no need for a rollback, since it is performed at the end of the test and also because we only retrieve entries in order to validate the invariant, we never modify an entry.

4.5 Validating the result of a call

What remains to be done is to check that each interface function returns an expected value. We want to obtain the information on what to expect as much as possible from the database content before the interface function is called and as little as possible from the abstract state, since we kept the abstract state as simple as possible.

To verify return values, we propose to implement a validator function for each interface function, which basically describes the conditions on the database that should be fulfilled in order to create a certain return value. This is easiest done by replacing the calls to the interface functions by local versions of them, which combine validation and actual interface call.

In our shop example, it is always possible to create a new customer and get a new identifier in return. We cannot possibly know which identifier will be returned, but there is no need for us to be aware of the precise value. Therefore, our local version of the `new_customer` function invokes the corresponding interface function for creating a user and matches the result with the expected value, in this case just checking its type. Failure to match the value results in a failing test case.

```
new_customer(Name)->
  Result = customer_facade:new_customer(Name),
  case Result of
   Id when is_integer(Id)-> Result;
   _                     -> exit(unexpected_value)
  end.
```

In this particular case the combination of validator and interface function only contains the interface function, since no additional conditions on the database are checked. If the actual value matches the expected value, i.e. is an integer, then we return that value, otherwise we raise an exception and the test fails.

However, more advanced interface functions, such as `place_order` demand a more complex validator function. There the business rules can be potentially affected by the state change caused by the interface call, so the state before the call needs to be inspected, and used to predict the expected result of the interface call. As far as interface function `place_order` is concerned, the related business rule, to be embodied by the corresponding validator, reads

Rule: "*featured products can only be ordered by gold customers*"

Intuitively, we infer from this that if a customer has 'gold' status, placing an order should always be a successful operation (no matter which products

are involved); and if the customer does not have the required status, the interface call will only be successful if the order does not contain a featured product. Additionally, we need to check that the product actually exists in the database, and that the customer `Id` is valid in order to perform the operation. For these rules we create SQL queries and transform the result of the query to a boolean validating each rule:

```
product_exists(Connection, PrCode) ->
 [] =/= db:process_query(Connection,
        "SELECT code FROM product WHERE code="
        ++ integer_to_list(PrCode)).

customer_exists(Connection, Id) ->
 [] =/= db:process_query(Connection,
        "SELECT id FROM customer WHERE id="
        ++ integer_to_list(Id)).

featured_product(Connection, PrCode) ->
 [PrCode] == db:process_query(Connection,
             "SELECT code FROM product WHERE code="
             ++ integer_to_list(PrCode) ++
             " AND code IN "++<FeatProdCodeList>).

gold_customer(Connection, Id) ->
 [Id] == db:process_query(Connection,
         "SELECT id FROM customer WHERE id="
         ++ integer_to_list(Id) ++
         " AND status='gold'").
```

Some of the above queries may well be equal to interface functions in the business logic layer. One could choose to re-use the interface functions here, but it has also value to separate concerns and specify the constraints in the test specification. In particular, if different people write tests specification and business logic layer, there is an extra possibility to identify failures.

Now we evaluate all constraints that we need before we execute the interface function. After that, we check whether the result we get is justified by the value of the conditions.

```
place_order(Id, PrCode) ->
 PE = product_exists(PrCode),
 CE = customer_exists(Id),
 FP = featured_product(PrCode),
```

```
 GC = gold_customer(Id),
 Result = order_facade:place_order(Id,PrCode),
 case Result of
    OrderId when is_integer(Result)
      and PE and CE and (not FP orelse GC) -> Result;
    {error,not_gold_customer}
      when FP and not GC                   -> Result;
    {error,non_existing_product}
      when not PE                          -> Result;
    {error,non_existing_customer}
      when not CE                          -> Result;
    _ -> exit(unexpected_value)
end.
```

We implement a case distinction on the obtained result instead of on the combination of conditions. On the one hand this reduces the amount of code, since we need only one alternative for each possible return value. On the other hand, we specify different from what we would do when implementing the business logic layer. If code is different, it is less likely that we make the same mistake. But even more importantly than the two reasons above, we do not have to bother about the (normally) unspecified implementations of dealing with double faults; in case both product and customer do not exist, we have no clue which of the errors is produced. By looking at the result, no matter which of the two error messages is produced, we accept it.

Note, first of all, that we validate the interface call prior to computing its result, since the interface call may result in a state change. Secondly, realise that the interface functions are (and should usually be) embedded in their own database transaction, making each call atomic and therefore avoiding any problem with running the system in a concurrent environment. This could apply to our testing environment as well, but neither the database used for the on-line shop, nor the databases for the case studies we considered supported nested transactions. One could argue that these kind of tests are dangerous, since the database may change between validating the state and calling the interface function. Therefore, it is essential to perform these tests sequentially, with just one client querying the database when validating and calling the sequence of interface calls. From our point of view, if all interface calls are indeed embedded in their own transactions, then this sequential testing is no real obstacle, since in a real situation we will not query the database to predict the results. Of course, there may be issues in the concurrent setting that cannot be discovered with this method, and other strategies such as load testing could be advised for testing those.

4.6 The QuickCheck property

Finally, we need to specify a property for QuickCheck to make this all work. This property states that random sequences of commands should be generated using the `command` generator described in Section 4.1 [6], and for each of those, the database invariant (in this case, the business rule as stated at the beginning of Section 2.2) is checked both before and after executing the testing sequence: if it holds in the resulting system state, the test passes.

```
prop_business_logic() ->
  ?FORALL(Cmds, commands(?MODULE),
    begin
      true = invariant(),
      {_,_,Res} = run_commands(?MODULE,Cmds),
      PostCondition = invariant(),
      clean_up(S),
      PostCondition and Res == ok
    end).
```

The reason to check whether the database invariant is fulfilled before the test starts is to be able to use the property also on databases that are already populated with some data.

5 Evaluating the method

This testing method we have just described has been used not only with the shop example that served as explanatory thread in this article, but also on two real case studies: on one of the main subsystems of the ARMISTICE risk management system, and on part of the interface of a financial system.

Both case studies are applications of many thousands of lines of code that are developed and used in a commercial environment. The databases used by these systems are different: for the first case study it was a relational database, whereas in the second case study it was a non-relational database developed for telecommunication applications.

5.1 ARMISTICE

As a first real life example for the method we have developed and described in this article, we have used the risk management application ARMISTICE, an information system that deals with the complex business domain of insurance management [6,10,11]. ARMISTICE has been in production for several years,

[6] The Erlang macro `?MODULE` represents the current module name.

after being tested by regular users during the last stages of development. Such testing process is common in software development, but it is hardly ever complete and exhaustive. The fact that an application has been running daily without major problems is just a weak empirical evidence of correctness.

In order to provide a greater degree of confidence, we decided to use QuickCheck to automatically generate random tests. In previous research, we have used QuickCheck to test ARMISTICE data types [3]. By using QuickCheck to test data types we successfully identified an error, even though the system had been in operation for such a long time. That error was eventually found connected to some obscure bug reports that had not been clarified so far. Now we have applied the method described in this paper to test the system's business logic.

Even though no actual errors were found by applying this testing method to the case study, the thousands of automatic tests that were successfully executed in a matter of minutes grant us a much greater level of confidence in the correctness of the business logic. As a side-effect, we now also have a formal specification of (part of) the business rules for this system, which simplifies future extensions and additions.

However, in order to further ensure that failure of the business logic implementation to enforce the constraints imposed by the business rules could be detected by this method, we purposely introduced errors in the production code. For instance, we removed specific parts of the code in which consistency was explicitly maintained, we ignored data conditions during function execution, and we implemented fictitious failing cases. We introduced these kinds of anomalies both on an individual basis and in combination, and with the method proposed in this paper we were able to generate test sequences that crashed due to the injected faults in both cases, and the QuickCheck tool provided smaller counterexample sequences of interface function invocations, which in a real situation would have improved error diagnosis and correction.

5.2 Financial System

A second real life example was the financial system of a large Swedish company. The database forms the core of their business handling and storing all customer transactions.

The method as presented in this article was applied to part of that system and a full report can be found in [15]. As a result of running randomly generated tests, several violations of the identified and validated business rules were detected. It was concluded that the applied methodology is suitable for applications using non relational, unnormalised databases. The study has revealed that the quality of elicited business rules depends on the understand-

ing of the application and the adequacy of database structure documentation obtained in the reverse engineering process. It was concluded that adopting the described approach can be used to identify potential violations of data constraints. It is important to note the methodology applied within the case study is not bound to a specific application or DBMS, and can be applied to other data-intensive applications.

6 Conclusions

In this paper, we have presented a testing method suitable for data-intensive applications. Its main guidelines summarise as follows:

- Formulate the **business rules as SQL queries** and write an **invariant function** that evaluates whether the data in the database respects them all.
- Write a **QuickCheck generator for sequences of interface calls** in which a minimum state is used (only to create consecutive operations on specific entities), covering both positive and negative testing.
- For each interface function, write a local checking function in which firstly a number of predicates are evaluated (conditions on the existing data). Secondly, the actual interface function is called. And thirdly, it is validated whether the **result of the real interface function corresponds to what could be expected** from the previously computed predicates.
- Create tests by generating **sequences of interface functions**, **validate** them via the local checking functions defined, and **check that the invariant holds after the execution** of the sequence.

Since the most relevant element in a data-intensive application is the database, we have identified the global state of the system with the state of the database, where the interface functions are the operations that can result in a state change. We abstract from the database content the minimum amount of information needed to conduct relevant and meaningful tests. We used QuickCheck to specify our test model as a finite state machine model. The different elements of our testing state machine are: a command generator that takes frequencies into account, a function to compute the next state, auxiliary data generators to automatically build proper interface function arguments, and finally a validator function to anticipate the result of command invocation.

In order to ensure business rule compliance we check SQL formalisations of business rules as invariants of the subject under test.

In our method we have assumed the RDBMS itself to be error free. We also assume that transactions are properly used to guarantee atomicity of the interface functions, such that testing in a concurrent setting is unnecessary for the kind of errors we are looking for. Of course, additional load testing may still explore the concurrency and distribution capabilities of the application. Last, we assume that unit tests have been performed for the system components, and this testing method is proposed at the system level testing.

One of the advantage of having a model for testing is that we can perform hundreds, thousands, or even hundreds of thousands of different tests, automatically generated and executed. If there is a change in the business rules, we can perform new tests only after reformulating a few SQL queries and properly updating the test invariants.

We have used and evaluated this testing method in industrial case studies with successful results in the capability of quickly and efficiently detect scenarios in which business rules are violated.

Acknowledgement

The authors would like to thank Víctor M. Gulías from the University of A Coruña for his support and for creating the possibility for this collaboration.

References

[1] *ARMISTICE: Advanced Risk Management Information System – Tracking Insurances, Claims and Exposures*, http://www.madsgroup.org/armistice/ (2002).

[2] Armstrong, J., R. Virding, C. Wikström and M. Williams, "Concurrent Programming in Erlang," Prentice-Hall, 1996, 2nd edition.

[3] Arts, T., L. M. Castro and J. Hughes, *Testing Erlang Data Types with Quviq QuickCheck*, in: *Proc. of the 7th ACM SIGPLAN Workshop on Erlang* (2008), pp. 1–8.

[4] Arts, T., J. Hughes, J. Johansson and U. Wiger, *Testing Telecoms Software with Quviq QuickCheck*, in: *Proc. of the 5th ACM SIGPLAN Workshop on Erlang* (2006), pp. 2–10.

[5] Bajec, M. and M. Krisper, *A methodology and tool support for managing business rules in organisations*, Information Systems **30** (2005), pp. 423–443.

[6] Cabrero, D., C. Abalde, C. Varela and L. M. Castro, *Armistice: An experience developing management software with erlang*, in: *Proc. of the 2nd ACM SIGPLAN Workshop on Erlang* (2003), pp. 23–28.

[7] Cesarini, F. and S. Thompson, "Erlang Programming," O'Reilly, 2009.

[8] Dietrich, J. and A. Paschke, *On the test-driven development and validation of business rules*, in: *Information Systems Technology and its Applications*, 2005, pp. 31–48.

[9] Emmi, M., R. Majumdar and K. Sen, *Dynamic test input generation for database applications*, in: *International Symposium on Software Testing and Analysis* (2007), pp. 151–162.

[10] Gulías, V. M., C. Abalde, L. M. Castro and C. Varela, *A new risk management approach deployed over a client/server distributed functional architecture*, in: *International Conference on Systems Engineering* (2005), pp. 370–375.

[11] Gulías, V. M., C. Abalde, L. M. Castro and C. Varela, *Formalisation of a functional risk management system*, in: *International Conference on Enterprise Information Systems* (2006), pp. 516–519.

[12] Halfond, W. and A. Orso, *Command-form coverage for testing database applications*, in: *International Conference on Automated Software Engineering* (2006), pp. 69–78.

[13] Hughes, J., *Quickcheck testing for fun and profit*, in: *International Symposium on Practical Aspects of Declarative Languages* (2007), pp. 1–32.

[14] *Junit*, http://www.junit.org (2008).

[15] Paladi, N. and T. Arts, *Model Based Testing of Data Constraints: Testing the Business Logic of a Mnesia Database Application with Quviq QuickCheck*, in: *Proc. of 8th ACM Sigplan Erlang workshop*, 2009, pp. 71–82.

[16] Willmor, D. and S. M. Embury, *Testing the implementation of business rules using intensional database tests*, in: *Proceedings of Testing: Academic & Industrial Conference on Practice And Research Techniques* (2006), pp. 115–126.